
ORACLE®


JavaOne™
ORACLE®

API Design with Java 8 Lambda and Streams

Stuart Marks
Twitter: @stuartmarks

Brian Goetz
Twitter: @briangoetz

Oracle Java Platform Group

JavaOne
ORACLE

Introduction

- Impact of new Java 8 language features on your APIs
  - Lambda
  - Streams
  - Optional
  - Default Methods
- When to use, when ***not*** to use, how to use effectively
- We're taking questions on Twitter – tweet with hashtag #JavaAPI

Lambda

Lambda

- Allows passing ***behavior*** through an API, not just values
  - concise, efficient means of expressing “code as data”
  - parameterizing with behavior (not just values and types)
  - this is a big new tool in the API design toolbox
- APIs previously used anonymous inner classes to pass code as data
  - create a new class, then a new instance
  - overall a roundabout way to express a bit of behavior
  - too clunky to use widely in APIs

Example: ThreadLocal

- Instance of the Template Method pattern to do lazy initialization
- Before Java 8, to provide an initial value
  - Subclass and override initialValue() method
  - initialValue() called at first get() call
  - value cached for subsequent get() calls
- Java 8: use lambda to “plug in” initialization function into the right place
  - no need for subclassing

ThreadLocal

```
// OLD

static ThreadLocal<Integer> threadId =
    new ThreadLocal<Integer>() {
        protected Integer initialValue() {
            return computeNextId();
        }
    };

// NEW

static ThreadLocal<Integer> threadId =
    ThreadLocal.withInitial(() -> computeNextId());
```

Example: Multi-Valued Map

- Task: maintain a map with multiple keys for each value

```
Map<Key, List<Value>>
```

- To add a (key, value) pair
  - first check to see if the key is present in the map
  - if it isn't
    - create an empty list
    - add the value to the list
    - put the key and list into the map
  - if the key is present
    - get the list
    - add the value to the list

Example: Multi-Valued Map

```
Map<Key, List<Value>> map = ... ;

// OLD

    List<Value> list = map.get(key);
    if (list == null) {
        list = new ArrayList<>();
        map.put(key, list);
    }
    list.add(newValue);

// NEW

    map.computeIfAbsent(key, k -> new ArrayList<>())
       .add(newValue);
```

Conditional Execution in Java 8

- Map.computeIfAbsent()
  - if key is absent, computes a value, puts it into map, returns it
  - if key is present, returns the value
- Advantages
  - encapsulates highly stylized code into the library
  - gives it a nice name
  - can be made atomic for concurrent maps

Example: Sorting Collections

- Existing sort methods
  - Collections.sort(List)
  - Collections.sort(List, Comparator)
- Common cases that should be supported by the library
  - sort by a field or property (sort by name, sort by age) using “key extractor” function
  - reversed-order sort
  - special handling for null (nulls-first, nulls-last)
  - multi-level sort (sort by last name, then by first name)
- Answer has historically been: “Provide your own Comparator”
  - but writing your own comparator is tedious and error-prone

How Many Sorting Methods to Provide?

sort()

sortReversed()

sortBy(extractor)

sortByReversed(extractor)

sortByInt(intExtractor)

sortByIntReversed(intExtractor)

sortByDouble(dblExtractor)

sortByDoubleReversed(dblExtractor)

sortNullsFirst()

sortNullsLast()

sortNested(extractor1, extractor2)

sortNestedIntObj(intExt1, ext2)

sortNestedObjInt(ext1, intExt2)

sortReversedNested(ext1, ext2)

sortNestedReversed(ext1, ext2)

...

Seems Like the Wrong Direction

- Adding sort method variations isn't working
  - combinatorial explosion of different methods
  - can try to minimize, but if one is missing, client is out of luck
  - "This is not the abstraction you are looking for"
- Time to step back and reconsider the problem

Think about Comparators Instead of Sorting

- A Comparator is just a function:
  - (T, T) => { < 0, 0, > 0 }
- Most Comparators are highly stylized code
  - complexity comes in when multiple cases are combined
  - this suggests a way to break things down and simplify them

Base Case: Comparator from Field Extractor

```
// some data class

    class Student {
        public String getLastName() { ... }
        public String getFirstName() { ... }
        public int getScore() { ... }
    }


    Comparator<Student> studentsByLastName =
        (s1, s2) -> s1.getLastName().compareTo(s2.getLastName())


    Comparator<Student> studentsByScore =
        (s1, s2) -> Integer.compare(s1.getScore(), s2.getScore())
```

Base Case: Comparator from Field Extractor

- Commonality
  - the same function is run on two objects, resulting in two Comparable values
  - these values are then compared
  - extract this into a static utility method

```
// NEW
(s1, s2) -> s1.getLastName().compareTo(s2.getLastName())
(s1, s2) -> Integer.compare(s1.getScore(), s2.getScore())

// NEW AND IMPROVED
Comparator.comparing(Student::getLastName)
Comparator.comparingInt(Student::getScore)
```

Creating Comparator Variants: Null Handling

```
Comparator<Student> studentsByFirstNameNullsFirst =
    (s1, s2) -> {
        String fn1 = s1.getFirstName();
        String fn2 = s2.getFirstName();
        if (fn1 == null)
            return (fn2 == null) ? 0 : -1;
        else
            return (fn2 == null) ? 1 : fn1.compareTo(fn2);
    };
```

Creating Comparator Variants: Two-Level Sorting

```
Comparator<Student> studentsByLastNameThenFirstName =
    (s1, s2) -> {
        int r = s1.getLastName().compareTo(s2.getLastName());
        if (r != 0)
            return r;
        else
            return s1.getFirstName().compareTo(s2.getFirstName());
    };
```

Two-Level Sorting *and* Null Handling

```
Comparator<Student> studentsByLastNameThenNullableFirstName =
    (s1, s2) -> {
        int r = s1.getLastName().compareTo(s2.getLastName());
        if (r != 0) {
            return r;
        } else {
            String fn1 = s1.getFirstName();
            String fn2 = s2.getFirstName();
            if (fn1 == null)
                return (fn2 == null) ? 0 : -1;
            else
                return (fn2 == null) ? 1 : fn1.compareTo(fn2);
        }
    };
```

Creating a Null-Handling Comparator

```
// function that null-specializes a comparator and returns a new comparator

static <T> Comparator<T> nullsFirst(Comparator<T> original) {
    return (t1, t2) -> {
        if (t1 == null)
            return (t2 == null) ? 0 : -1;
        else
            return (t2 == null) ? 1 : original.compare(t1, t2);
        };
}

// example

Comparator<Student> studentsByFirstNameNullsFirst =
    Comparator.comparing(Students::getFirstName, nullsFirst(naturalOrder()));
```

Creating Comparator Variants: Two-Level Sorting

```
// default method in Comparator interface

default Comparator<T> thenComparing(Comparator<T> other) {
    return (t1, t2) -> {
        int res = this.compare(t1, t2);
        return (res != 0) ? res : other.compare(t1, t2);
    };
}

// example

Comparator<Student> studentsByLastNameThenFirstName =
    Comparator.comparing(Student::getLastName)
              .thenComparing(Student::getFirstName);
```

Comparator Example

```
students.sort((s1, s2) -> {                                          // OLD
    int r = s1.getLastName().compareTo(s2.getLastName());
    if (r != 0)
        return r;
    String f1 = s1.getFirstName();
    String f2 = s2.getFirstName();
    if (f1 == null) {
        return f2 == null ? 0 : -1;
    } else {
        return f2 == null ? 1 : f1.compareTo(f2);
    }
});

                                                                     // NEW
students.sort(comparing(Student::getLastName)
          .thenComparing(Student::getFirstName, nullsFirst(naturalOrder())));
```

Lessons from Comparator API

- Some APIs have combinatorial explosion of complexity
  - look for proliferation of method variations (e.g., sort methods)
  - look for long parameter lists, with many optional parameters
  - look for lots of overloads with different variations of parameters
- Large number of variations comes from combinations of smaller features
- Break down the problem into smaller features that can be composed
  - write higher order functions to do the composition
  - allow user to plug in logic using lambdas
  - combination of static factories and default methods

Streams

Adding Streams to APIs

- Early Java 8 effort – "lambdafication"
  - many objects are conceptually containers of other objects
  - easy step: add forEach() method on them
- But also want to transform, filter, sort, etc.

forEach	forEachSorted
forEachFiltered	forEachFilteredSorted
forEachMapped	forEachMappedSorted
forEachFilteredMapped	forEachFilteredMappedSorted

- Sound familiar?

Adding Streams to APIs

- "Lambdafication" quickly turned to "Streamification"
  - adding a single stream() method opens up full range of stream functionality
  - many conceptually aggregate objects can return collections
  - should they return a stream or a collection or both?
- Mostly, doesn't matter
  - easy for caller to convert a stream into a collection and vice versa

Stream vs. Collection

- Stream instead of Collection
  - creating the collection is expensive
  - cheaper to produce elements lazily on demand
  - caller needs only a subset of the elements (filter, findFirst), can short-circuit
  - avoids creating defensive copies
  - returned stream can be infinite
- Collection instead of Stream
  - snapshot semantics
  - caller needs to traverse multiple times
  - or in different directions

How to Return a Stream

- If you have zero elements
  - Stream.empty()
- If you have a fixed number of elements
  - Stream.of(e1, e2, e3, ...)
- If you have a collection
  - just call stream()
- If you have an array
  - call Arrays.stream(array)

Create a Stream from an Iterator

```
// if size unknown

StreamSupport.stream(
    Spliterators.spliteratorUnknownSize(iterator, 0), false)

// if size is known

StreamSupport.stream(
    Spliterators.spliterator(iterator, size, 0), false)
```

Create a Spliterator, then a Stream

- Create subclass of Spliterators.AbstractSpliterator
  - only one method required: tryAdvance()

```
boolean tryAdvance(Consumer<Object> consumer) {
    Object obj = getTheNextObject();
    if (obj == null)
        return false;
    consumer.accept(obj);
    return true;
}
```

  - for improved sequential performance, implement forEachRemaining()
  - for better parallel scaling, implement trySplit()

Create a Spliterator, then a Stream

- Once you have a spliterator, call
  - StreamSupport.stream(spliterator, isParallel)
- Consider also primitive specializations for int, long, double

Spectrum of Stream-Returning Techniques

- Create from Iterator
  - Spliterators.spliteratorUnknownSize
  - Spliterators.spliterator
- Create from Spliterator
  - AbstractSpliterator.tryAdvance
  - AbstractSpliterator.forEachRemaining
  - AbstractSpliterator.trySplit
- Later ones are more effort, but offer improved performance

Why Spliterator?

- Iterator
  - two method calls per element traversed: hasNext() and next()
  - often interact in subtle ways
    - hasNext() must cache value for next() to return
  - need to guard against unusual call order
    - e.g., next() called twice in succession
- Spliterator
  - one method per element: tryAdvance()
  - a better iterator than Iterator, even for sequential execution
  - adds splitting abstraction for parallelism

Optional

The Primary Use of Optional

Optional is intended to provide a ***limited*** mechanism for library method ***return types*** where there is a clear need to represent "no result," and where using null for that is overwhelmingly ***likely to cause errors***.

When To Use Optional

- Use as method return value, when absence of a value is an ***expected*** result
  - as opposed to an exceptional result
  - example: findFirst() or similar method
  - allows caller to deal with absence of value without checking for null
  - allows convenient method chaining
- A method returning Optional should **NEVER** return null!
- Terminology note: prefer “empty Optional” over “Optional containing null”

When To Use Optional

- Method chaining
  - returning an Optional allows caller to chain methods safely
  - orElse() – returns value if present, else substitutes a default value
    - NOTE: avoid orElse(null) if possible
  - orElseGet() – returns value if present, else calls a lambda to generate the value
  - orElseThrow() – returns value if present, else throws the given exception
  - get() – returns a value if present, otherwise throws NoSuchElementException
    - WARNING: use get() only if you can ***prove*** the value is always present!

Examples

```
Optional<String> match = words.stream()
                              .filter(word -> word.startsWith("A"))
                              .findFirst();

System.out.println(match.orElse("not found"));

System.out.println(match.orElseGet(() -> getNotFoundMessage()));
```

When Not To Use Optional

- It's very tempting to use Optional in other contexts
  - method arguments
  - object fields
  - in a collection
- It ***seems*** like these techniques ought to work
  - end up cluttering and obscuring code unnecessarily

When Not To Use Optional

- It is ***not*** a goal of Optional to get rid of nulls everywhere
- Yes! Sometimes it's ok to use null
  - a private field with null as a sentinel can easily be verified correct
  - as a method argument – you check your arguments, right?
- Returning a collection, array, or stream
  - don't return Optional<Collection<T>> or Optional<Object[]> or Optional<Stream<T>>
  - don't return null
  - instead, return an empty collection, array, or stream

Method Chaining is Cool, But...

```
// BAD

String process(String s) {
    return Optional.ofNullable(s).orElseGet(this::getDefault);
}


// GOOD

String process(String s) {
    return (s != null) ? s : getDefault();
}
```

Summary of Optional

- Focus on using Optional as a return value
  - where search or computation might not return a result
  - and where returning null is likely to cause errors
- Resist temptation to apply Optional elsewhere
  - it's not necessarily *wrong,* but it's unlikely to be useful
  - misuse of Optional has led to the invention of several new code smells
- Optional works well for specific cases
  - don't overdo it!

Default Methods

Primary Use Case: Evolving an Existing Interface

- Before Java 8, adding a method to an interface could result in AbstractMethodError
  - so basically it was never done
- Default methods are interface methods plus a fallback implementation
- Default methods are ordinary virtual methods and can be overridden

Example Default Method in Interface

- Iterable.forEach

```
default void forEach(Consumer<? super T> action) {
    Objects.requireNonNull(action);
    for (T t : this) {
        action.accept(t);
    }
}
```

- Implemented only in terms of statics, ***this***, and parameters

Default Methods: Secondary Use Cases

- When the method is optional
  - example: Iterator.remove()

```
default void remove() {
    throw new UnsupportedOperationException("remove");
}
```

- Convenience method, not necessary to be overridden
  - example: Comparator.reversed()

```
default Comparator<T> reversed() {
    return (t1, t2) -> this.compare(t2, t1);
}
```

Default Methods vs Abstract Classes

- Abstract classes are obsolete now that we have default methods, right?
- No! Classes still have the following that interfaces do not:
  - state (fields)
  - constructors (allowing control over instance creation)
  - protected methods
    - allow communication with subclasses as distinct from callers
- Before adding a default method, ask whether it's useful to callers
  - interface methods are all public
  - don't use default methods for sharing code among implementors
  - if it's only useful to subclassers, maybe you should use an abstract class instead

Default Method Tradeoffs

- Incompatibility risks
  - possible name collisions, e.g., List.sort()
  - fragile superclass problem
    - same issue that has always existed for classes
    - arguably riskier for interfaces, since they're more widely subclassed
- Works well for intended use
  - if applied judiciously
  - if applied outside intended use, results are often unsatisfactory
  - misuse of default methods is another generator of new code smells

Summary

- Lambda
- Streams
- Optional
- Default methods

Safe Harbor Statement

The preceding is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.


JavaOne™
ORACLE®


ORACLE®